函数拟合

2253102 鲁浩宇

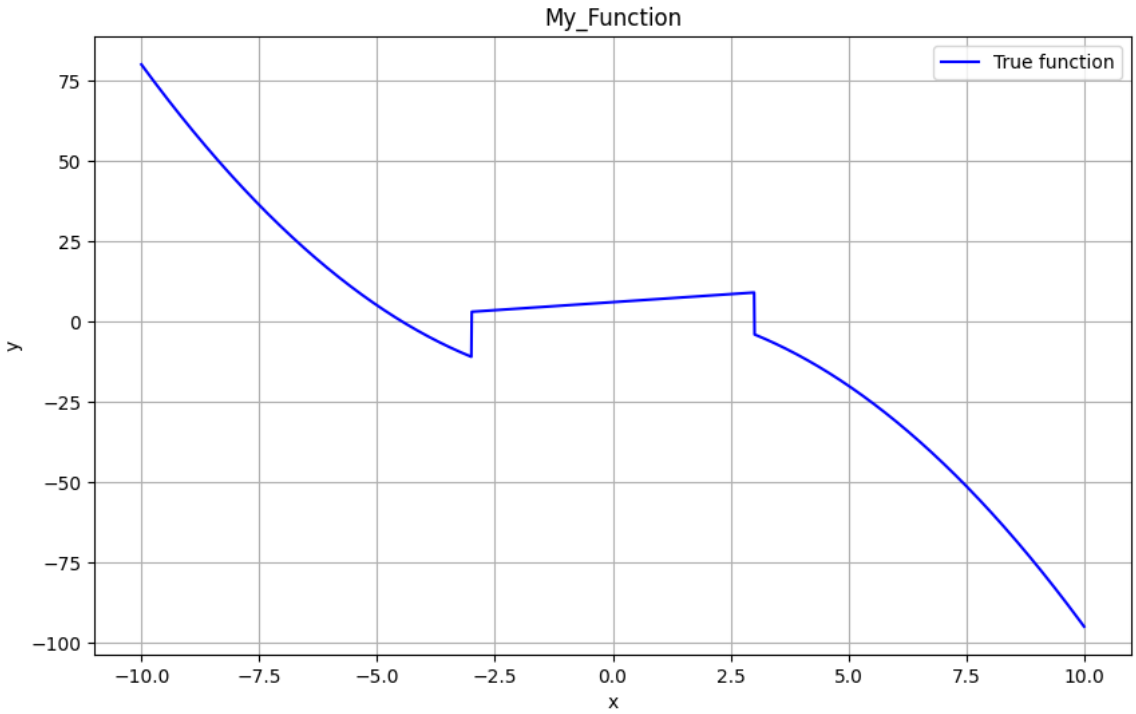
函数定义

```python
def my_func(x):
    '''要逼近的目标函数'''
    if x < -3:
        return x ** 2 - 20  # x<-3时为二次函数
    elif x < 3:
        return x + 6  # -3≤x<3时为线性函数
    else:
        return -x**2 + 5  # x≥3时为二次函数
```

a. **分段逻辑**：
   - 第一条件 `x < -3` 对应二次函数分支。
   - 第二条件 `-3 ≤ x < 3` 对应线性函数分支。
   - 剩余情况 `x ≥ 3` 对应另一个二次函数分支。

b. **边界处理**：
   - 在 `x = -3` 时，`x + 6` 和 `x^2 - 20` 的值均为 `3`（连续）。
   - 在 `x = 3` 时，`x + 6` 和 `-x^2 + 5` 的值均为 `9` 和 `-4`（不连续）。

c. **函数图像**：



数据采集

```python
def get_data(num, batch_size):
    '''生成训练数据'''
    data_x = []
    data_y = []
    for _ in range(num):
        # 生成-10到10之间的随机x值
        x = np.array([random.uniform(-10, 10) for _ in range(batch_size)])
        # 计算对应的y值（使用目标函数）
        y = np.array([my_func(item) for item in x])
        # 调整形状为(1, batch_size)
        x = np.expand_dims(x, axis=0)
        y = np.expand_dims(y, axis=0)
        data_y.append(y)
        data_x.append(x)
    return data_x, data_y
```

a. **数据生成函数**

- 使用 `get_data(num, batch_size)` 生成数据，其中：
- `num`：生成的 batch 数量。
- `batch_size`：每个 batch 的样本数。

b. **输入数据 (`x`)**：

- 在 `[-10, 10]` 范围内均匀采样，保证数据覆盖整个目标区间。
- 使用 `random.uniform(-10, 10)` 生成随机浮点数。

c. **输出数据 (`y`)**:

- 对每个 `x` 计算 `my_func(x)` ，得到对应的 `y` 值。
- 函数 `my_func(x)` 是分段定义的:
  - x<−3x<−3：y=x2−20y=x2−20（二次函数）
  - −3⩽x<3−3⩽x<3：y=x+6y=x+6（线性函数）
  - x⩾3x⩾3：y=−x2+5y=−x2+5（二次函数）

d. **数据形状调整**:

- 每个 `x` 和 `y` 被调整为 `(1, batch_size)` 的形状，便于后续批处理训练。

模型定义

```python
class myModel:
    """两层ReLU神经网络模型"""

    def __init__(self):
        # 初始化权重和偏置（随机正态分布）
        self.W1 = np.random.normal(size=[1, 60])  # 输入层到隐藏层的权重
        self.W2 = np.random.normal(size=[60, 1])  # 隐藏层到输出层的权重
        self.b1 = np.random.normal(size=[60])  # 隐藏层偏置
        self.b2 = np.random.normal(size=[1])  # 输出层偏置

        # 初始化各层运算
        self.mul_h1 = Matmul()  # 第一个矩阵乘法层
        self.mul_h2 = Matmul()  # 第二个矩阵乘法层
        self.add_h1 = Add()  # 第一个加法层（隐藏层）
        self.add_h2 = Add()  # 第二个加法层（输出层）
        self.relu = Relu()  # ReLU激活层

    def forward(self, x):
        '''前向传播'''
        # 第一层：x @ W1 + b1
        self.h1 = self.mul_h1.forward(x, self.W1)
        self.h1_add = self.add_h1.forward(self.h1, self.b1)

        # ReLU激活
        self.h1_relu = self.relu.forward(self.h1_add)

        # 第二层：h1_relu @ W2 + b2
```

```
        self.h2 = self.mul_h2.forward(self.h1_relu, self.W2)
        self.h2_add = self.add_h2.forward(self.h2, self.b2)

    def backward(self, label):
        '''反向传播（自动微分）'''
        # 输出层加法反向传播
        self.h2_add_grad, self.b2_grad = self.add_h2.backward(label)

        # 输出层矩阵乘法反向传播
        self.h2_grad, self.W2_grad = self.mul_h2.backward(self.h2_add_grad)

        # ReLU反向传播
        self.h1_relu_grad = self.relu.backward(self.h2_grad)

        # 隐藏层加法反向传播
        self.h1_add_grad, self.b1_grad =
self.add_h1.backward(self.h1_relu_grad)

        # 隐藏层矩阵乘法反向传播
        self.h1_grad, self.W1_grad = self.mul_h1.backward(self.h1_add_grad)
```

a. **模型概述**

`myModel` 是一个 **双层全连接神经网络**，包含：

- **输入层 → 隐藏层（60个神经元） → ReLU激活函数 → 输出层**
- 采用 **矩阵运算（Matmul）** 和 **加法运算（Add）** 实现前向传播
- 支持 **自动微分（Autograd）**，通过反向传播计算梯度

b. **模型结构**

(1) 初始化（`init`）

- **权重（W1, W2）** 和 **偏置（b1, b2）** 使用 **正态分布随机初始化**：
  - `W1`：输入层 → 隐藏层权重，形状 `[输入维度, 60]`
  - `W2`：隐藏层 → 输出层权重，形状 `[60, 输出维度]`
  - `b1`：隐藏层偏置，形状 `[60]`
  - `b2`：输出层偏置，形状 `[输出维度]`
- **计算层组件**：
  - `mul_h1`, `mul_h2`：矩阵乘法（`Matmul`）
  - `add_h1`, `add_h2`：加法运算（`Add`）
  - `relu`：ReLU激活函数（`Relu`）

(2) 前向传播（`forward`）

第一层计算（输入 → 隐藏层）：

线性变换：`h1 = X · W1`（矩阵乘法）

加偏置：`h1_add = h1 + b1`

ReLU激活：`h1_relu = ReLU(h1_add)`

第二层计算（隐藏层 → 输出层）：

线性变换：`h2 = h1_relu · W2`

加偏置：`h2_add = h2 + b2`（最终输出）

(3) 反向传播（`backward`）

输出层梯度计算：

加法层梯度：`add_h2.backward(损失梯度)` → 得到 `b2` 的梯度

矩阵乘法梯度：`mul_h2.backward(输出层梯度)` → 得到 `W2` 的梯度

隐藏层梯度计算：

ReLU梯度：`relu.backward(隐藏层梯度)`

加法层梯度：`add_h1.backward(ReLU梯度)` → 得到 `b1` 的梯度

矩阵乘法梯度：`mul_h1.backward(加法梯度)` → 得到 `W1` 的梯度

拟合效果